



# School of Computer Science & Engineering

## **COMP3891/9283 Extended Operating Systems**

2025 T2 Week 09

## **Object Capabilities**

Gernot Heiser

# Copyright Notice

**These slides are distributed under the Creative Commons Attribution 4.0 International (CC BY 4.0) License**

- You are free:
  - to share—to copy, distribute and transmit the work
  - to remix—to adapt the work
- under the following conditions:
  - **Attribution:** You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:

*“Courtesy of Gernot Heiser, UNSW Sydney”*

The complete license text can be found at  
<http://creativecommons.org/licenses/by/4.0/legalcode>

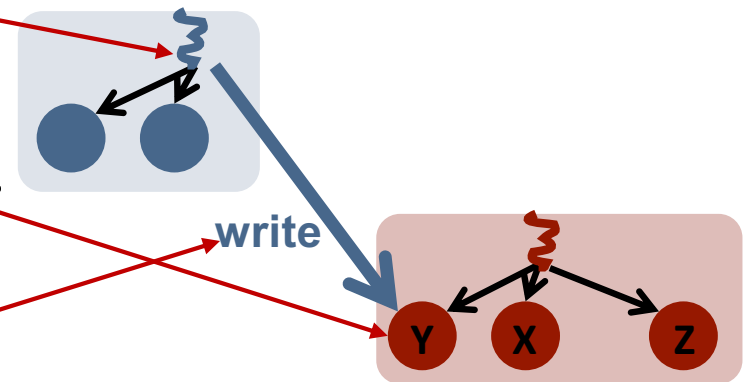
# Learning Outcomes

- Understanding of object capabilities as a fine-grained access control model
- Understanding of the pros and cons vs access-control lists
- Understanding of implementation approaches

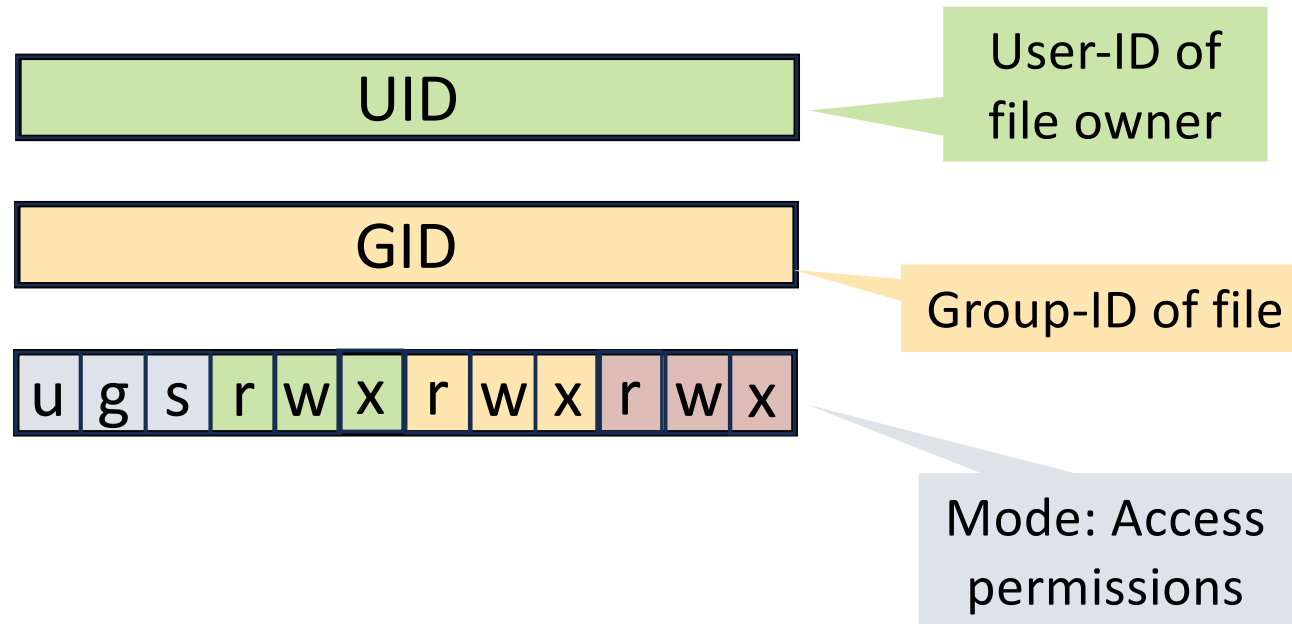
# Access Control

## Who can access **what** in which **ways**

- The “who” are called **subjects** (or **agents**)
  - e.g. users, processes etc.
- The “what” are called **objects**
  - e.g. individual files, sockets, processes etc.
  - includes all subjects!
- The “ways” are called **permissions**
  - e.g. read, write, execute etc.
  - are usually specific to the kind of object
  - include those meta-permissions that allow modification of the protection state
    - e.g. own

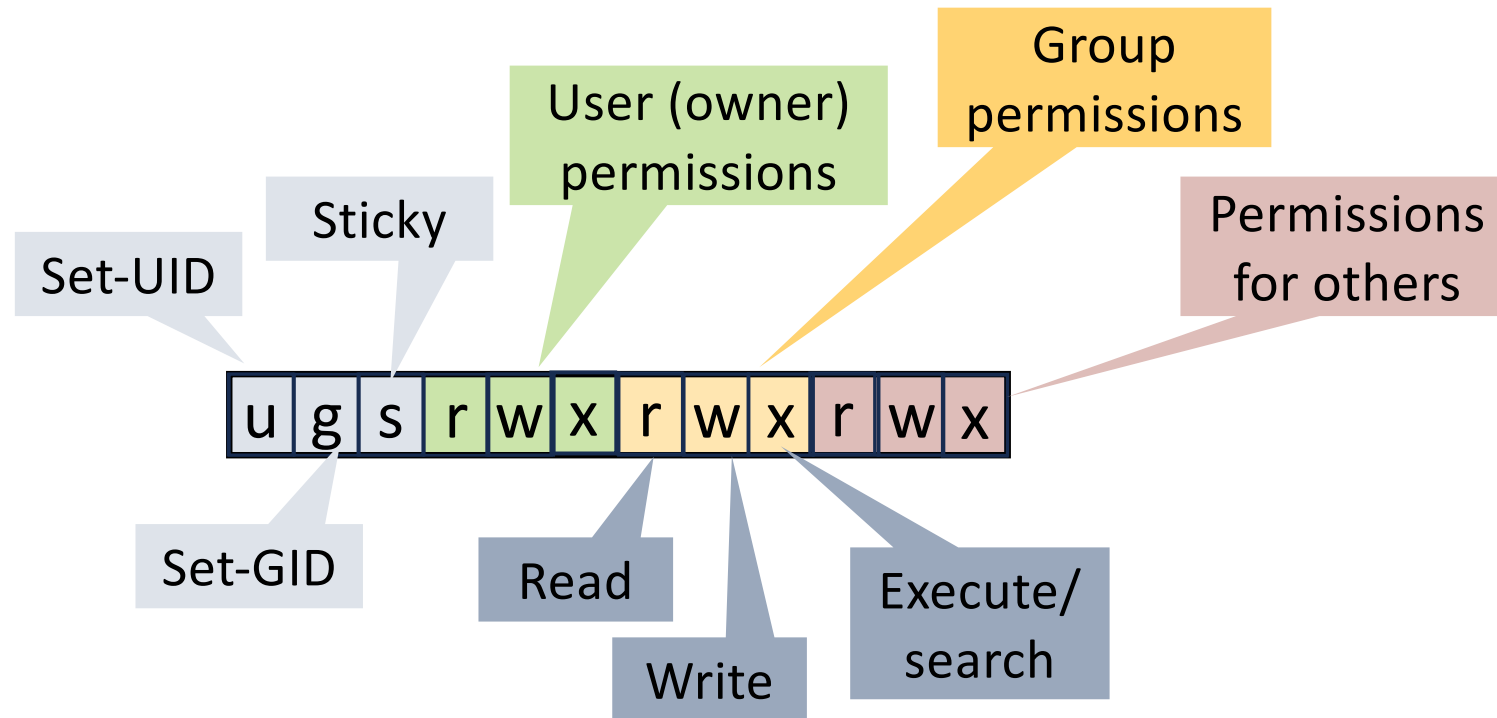


# Unix File Access Control: User/Group/Mode



- Identify file's owner and group
- Specify access rights
- Stored in i-node

# Unix File Access Control: Mode Word



for  $a \in \{rwx\}$ :  $a$  is allowed iff

$\text{subject.UID} == 0 \vee$

$(\text{subject.UID} == \text{file.UID} \wedge \text{u\_perm}.a) \vee$

$(\text{subject.UID} \in \text{group}(\text{file.GID}) \wedge \text{g\_perm}.a) \vee$

$\text{o\_perm}.a$

# More General: Access Control Lists (ACLs)

UID or GID

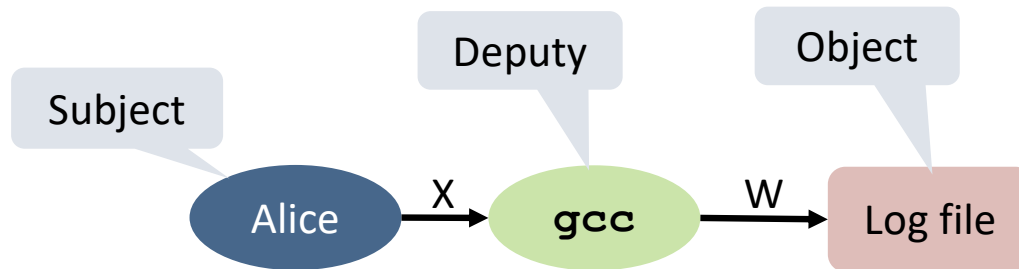
Can be  
negative right!

Ordered list of (id,perm) pairs

```
bool allowed (UID subj, ACCESS a, ACL acl) {  
    if (subj == 0) return TRUE;  
    for item in acl do {  
        if (subj ∈ item.id) {  
            if (a ∈ item.perm) return TRUE;  
            if (a ∈ !item.perm) return FALSE;  
        }  
    }  
    return FALSE;  
}
```

- Unix mode word is a compressed ACL
- Linux, BSD now offer full ACLs as well

# The Confused Deputy



## Unix:

- Log file is group **admin**
- Alice not member of **admin**
- gcc is set-GID **admin**

```
alice$ gcc -o Log_file source.c
```

```
static char* log = "/var/gcc/log";
int gcc (char *src, *dest) {
    int s = open (src, RDONLY );
    int l = open (log, APPEND);
    int d = open (dest, WRONLY);
    ...
    write (dest, ...);
}
```

Clobber log!

- ACLs separate naming and permissions
- Deputy depends on *ambient authority*: Uses own privileges for access

*Confused deputy is inherent problem of ACLs!*



# Protection State: Access-Control Matrix

Defines system's protection state at a particular time instance [Lampson '71]

Subjects are also objects

	Obj 1	Obj 2	Obj 3	Subj 2
Subj 1	R	RW		send
Subj 2		RX		control
Subj 3	RW		RWX own	recv

# Representing Protection State

Storing full matrix is infeasible

- huge but sparse
- highly dynamic

Obj 1	
Subj1:	R
Subj3:	
RW	

	Obj 1	Obj 2	Obj 3	Subj 2
Subj 1	R	RW		send
Subj 2		RX		control
Subj 3	RW		RWX own	recv

Columns are ACLs!

# Representing Protection State

```
bool allowed (ACCESS a, Cap c) {  
    return a ∈ c.perm;  
}
```

How about rows?

	Obj 1	Obj 2	Obj 3	Subj 2
Subj 1	R	RW		send
Subj 2		RX		control
Subj 3	RW		RWX own	recv

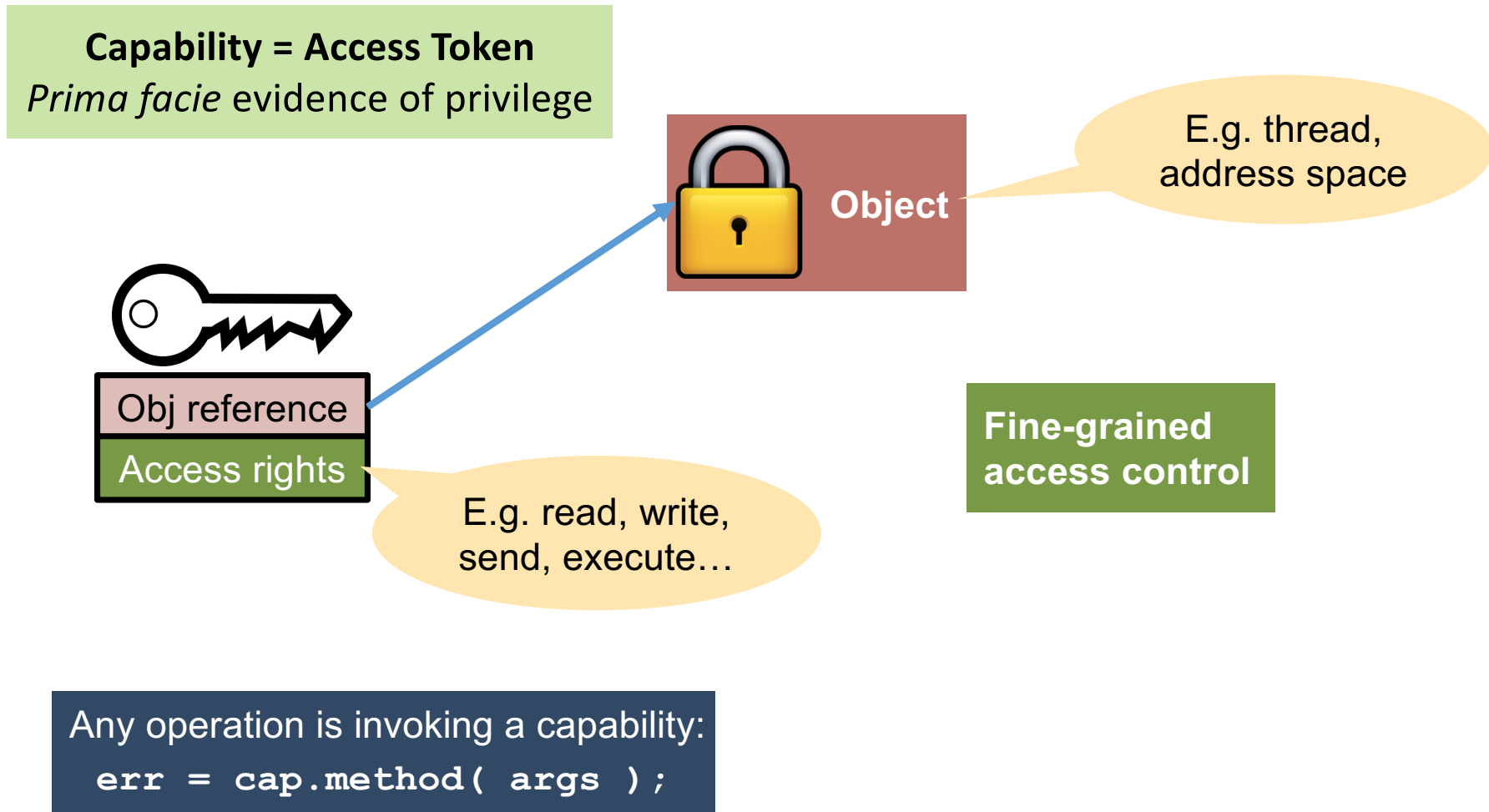
Set of rights a subject has –  
the subject's *protection domain*

“Object capability”

Subj 3  
Obj1: RW  
Obj3: RWX, own  
Subj2: recv

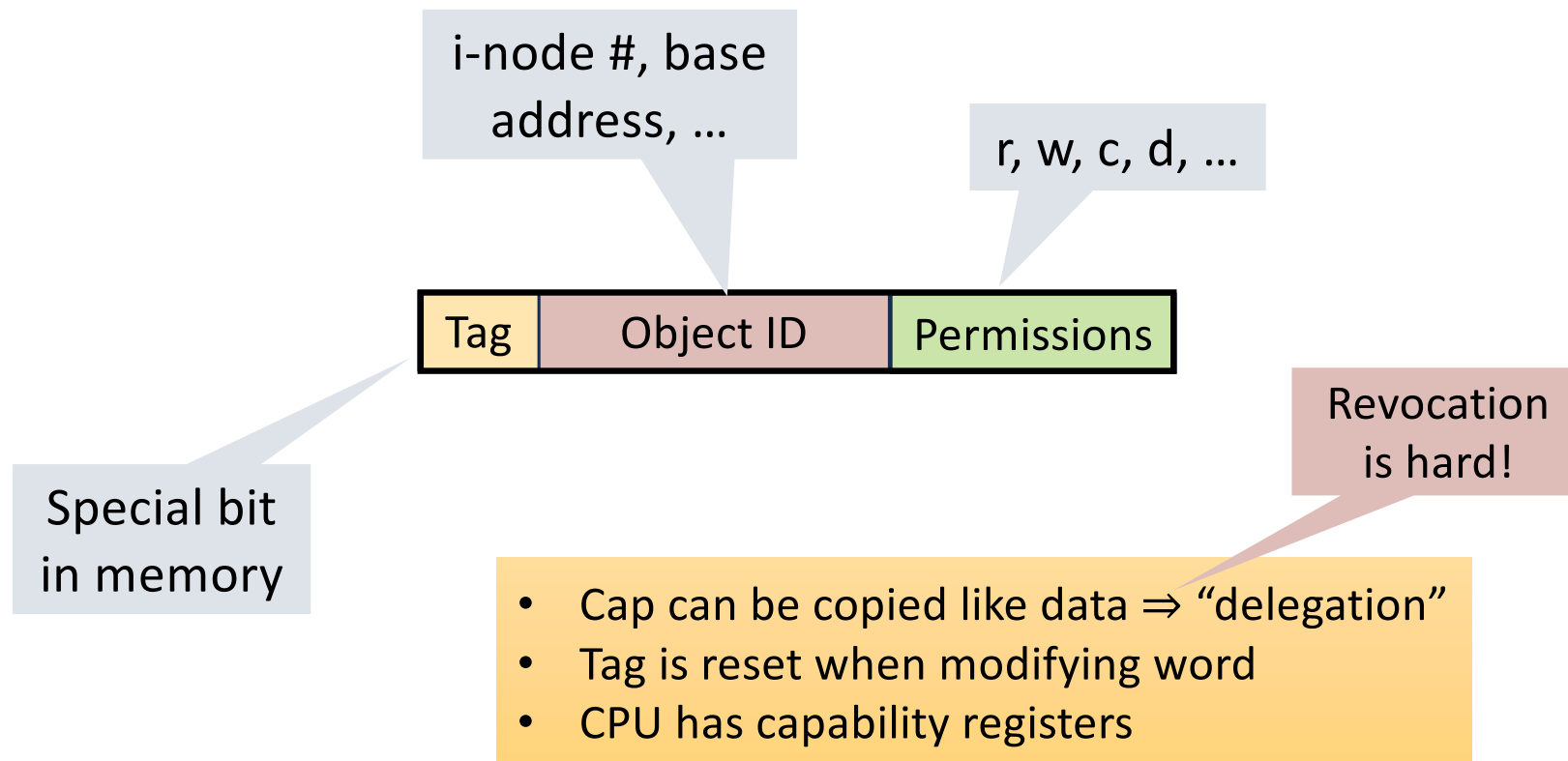
Represented as a  
*capability list* (Clist)

# (Object) Capabilities (aka Ocaps, Caps)



# Implementing Ocaps: Hardware

Recently  
revived: CHERI



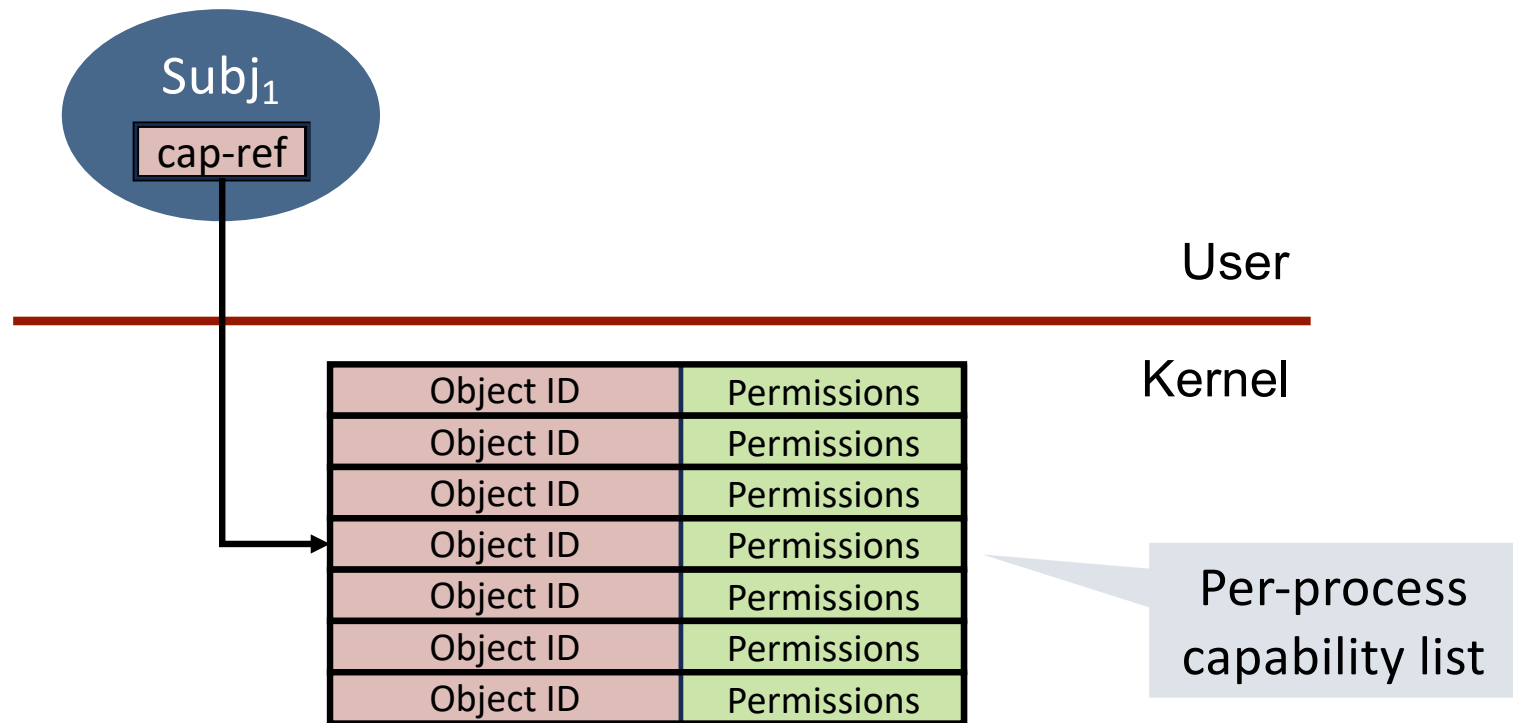
# Implementing Ocaps: Software-Usermode



Revocation  
is hard!

- Cap can be copied like data  $\Rightarrow$  delegation
- Signature mismatch when modifying cap
- OS has object table,  
holds signatures and permissions

# Implementing Ocaps: Software-Kernel



- Delegation is system call
- Revocation is easy

# Confused Deputy With Object Capabilities



```
alice$ gcc -o Log_file source.c
```

Invalid cap

```
static cap_t log = <cap>;
int gcc (cap_t src, dest) {
    fd_t s = open (src, RDONLY );
    fd_t l = open (log, APPEND);
    df_t d = open (dest, WRONLY);
    ...
    write (d, ...);
}
```

Open fails!

## Object capability (Ocaps) system:

- gcc holds **w** cap for log file
- Alice holds **r** cap for source, **w** cap for destination
- Alice holds no cap for log file

- Caps are both names and permissions
- Presented explicitly, not ambient
- Can't name object if don't have access!



# How About Linux Capabilities?

Capabilities(7)

Miscellaneous Information Manual

Capabilities(7)

## NAME

`capabilities` - overview of Linux capabilities

## DESCRIPTION

For the purpose of performing permission checks, traditional UNIX implementations distinguish two categories of processes: privileged processes (whose effective user ID is 0, referred to as superuser or root), and unprivileged processes (whose effective UID is nonzero). Privileged processes bypass all kernel permission checks, while unprivileged processes are subject to full permission checking based on the process's credentials (usually: effective UID, effective GID, and supplementary group list).

Starting with Linux 2.2, Linux divides the privileges traditionally associated with superuser into distinct units, known as capabilities, which can be independently enabled and disabled. Capabilities are a per-thread attribute.

# How About Linux Capabilities?

Linux “capabilities”  
restrict *system calls* a  
root process can invoke

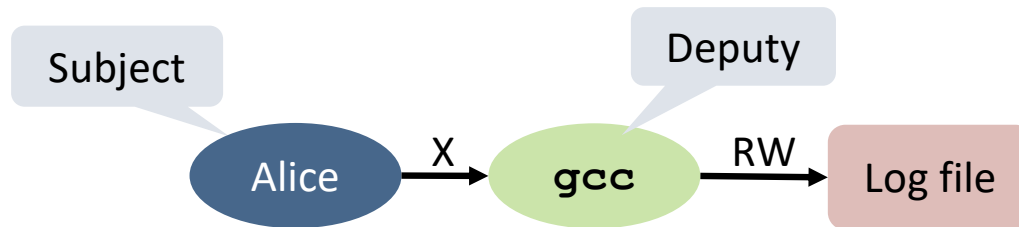
Linux “capabilities” aren’t Ocaps:  
Limit access to system calls, not objects

“Capability lists” are  
process attributes

```
bool allowed (UID subj, ACCESS a, ACL acl) {  
    if (subj == 0) return TRUE;  
    for item in acl do {  
        if (subj ∈ item.id) {  
            if (a ∈ item.perm) return TRUE;  
            if (a ∈ !item.perm) return FALSE;  
        }  
    }  
    return FALSE;  
}
```

syscall ∈ caplist;

# Confused Deputy With Linux Capabilities



```
alice$ gcc -o Log_file source.c
```

```
static char* log = "/var/gcc/log";
int gcc (char *src, *dest) {
    int s = open (src, RDONLY );
    int l = open (log, APPEND);
    int d = open (dest, WRONLY);
    ...
    write (dest, ...);
}
```

Still uses ambient authority!

## Unix:

- Log file is group **admin**
- Alice not member of **admin**
- gcc is set-GID **admin**

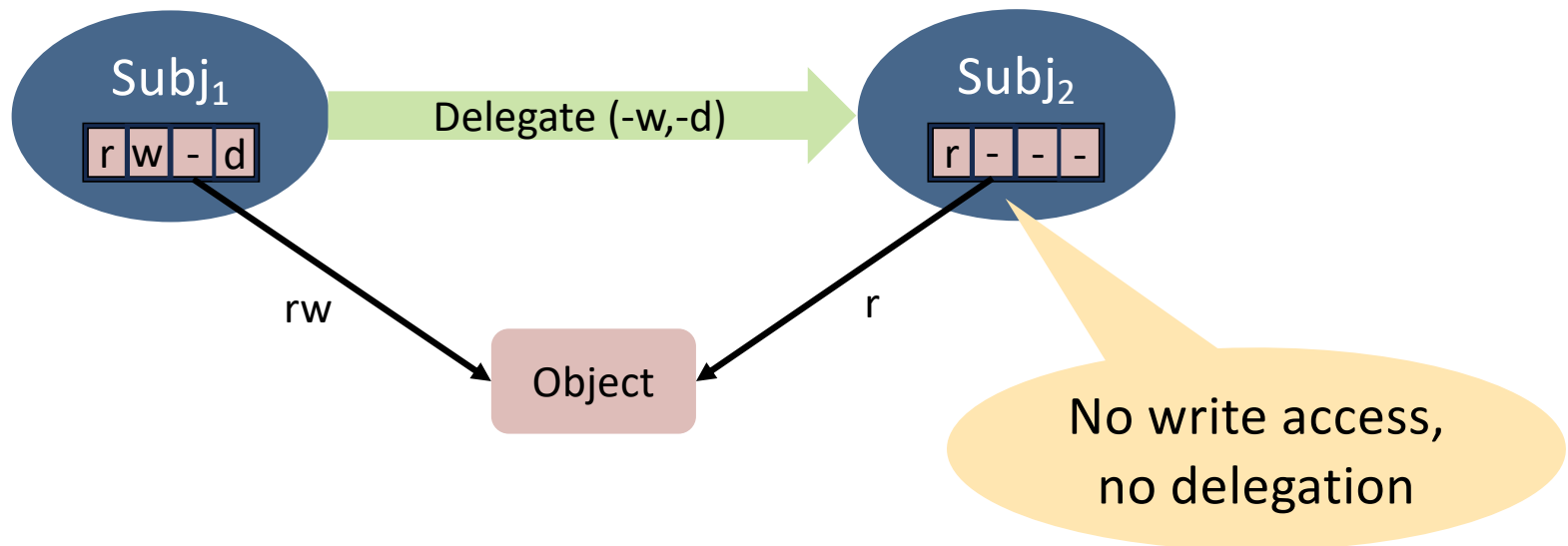
- gcc doesn't execute as root
- Needs w access to log file

Linux "capabilities" do nothing to prevent the deputy from being confused!

# Delegating Access

## Privilege conveyed by cap (eg files):

- Read (r)
- Write (w)
- Execute (x)
- *Delegate* (d)



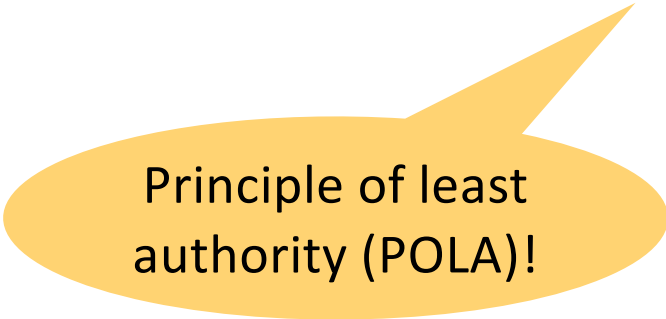
# ACLs vs Ocaps

## Access Control Lists

- Access based on identity
- Course-grained (by subj-ID)
- Objects referenced by name
- Ambient authority
- Delegation course-grained
  - setuid/setgid
- Large default access set

## Object capabilities

- Access based on holding cap
- Fine-grained (per object)
- Object referenced by Ocap
- Explicit cap presentation
- Delegation fine-grained
  - per object per subject
- No default access



Principle of least authority (POLA)!